14th International Workshop on OpenCL and SYCL

IWOCL 2026

AdaptiveCpp Portable CUDA: A SYCL-Compatible CUDA Compiler for CPUs and GPUs from Multiple Vendors.

Aksel Alpay, Universität Heidelberg

Aksel Alpay, Vincent Heuveline, Universität Heidelberg

May 6-8, 2026 | Heilbronn, Germany | iwocl.org

Motivation: Too Many Variables

The Problem: Implementer Perspective

- When comparing CUDA to SYCL, we typically change multiple variables:
  - Input Code
  - Programming model
  - Compiler
- Switching programming models, code **and** compilers obscures origin of performance deltas
- Is there a way to keep variables constant for fairer comparisons?

Motivation: The CUDA Portability Challenge

The Problem: User Perspective

- CUDA dominates existing GPU codebases
- Porting is required before using new programming models
- Existing tools (e.g. SYCLomatic) require full application porting
- Forces developers to commit to a single programming model
  - Existing CUDA code may be very complex, or highly optimized
  - Additional performance or correctness validation costs

What We Want

Requirements

- Compile CUDA/HIP code **as-is**
- Run on CPUs, NVIDIA GPUs, AMD GPUs, Intel GPUs, …
- Seamless interoperability with SYCL
- Iterative or partial porting capability – without sacrificing portability
- Compare SYCL and CUDA code with **same** compiler

Our Approach: AdaptiveCpp Portable CUDA (PCUDA)

Key Innovation

PCUDA is a compiler and runtime that compiles CUDA/HIP code and runs it on heterogeneous hardware, with full interoperability with SYCL.

Benefits

- Compile CUDA/HIP directly
- No porting required to start
- Mix CUDA and SYCL in same binary
- Same compiler for fair comparisons

Targets

- Any LLVM-supported CPU
- NVIDIA GPUs
- AMD GPUs
- Intel GPUs
- Apple GPUs
- Others

Background: Heterogeneous Programming Models

UNIVERSITÄT HEIDELBERG
ZUKUNFT SEIT 1386

SYCL (Khronos Standard)

- SPMD-style kernels as C++ lambdas
- `parallel_for` for kernel submission
- No annotations to mark device code needed
- Library-only mode possible for CPUs

CUDA / HIP

- Global functions with `__global__` attribute
- Triple chevron kernel launch: `<<<...>>>`
- Needs `__host__`, `__device__` attributes
- Needs dedicated compiler frontend

Common Ground

Both use work items (threads) grouped in work groups (blocks), with synchronization within groups.

Background: AdaptiveCpp

- Open-source heterogeneous compiler and runtime stack
- `https://github.com/adaptivecpp/adaptivecpp`
- Supports multiple programming models: SYCL, C++ standard parallelism offloading (stdpar), **now PCUDA**
- Highly portable via mature, native backends: OpenMP, CUDA, HIP, OpenCL+SPIR-V, Level Zero + SPIR-V
- **New** backends:
  - Metal (thanks Alexey Ozeritskiy!)
  - Vulkan (experimental, not yet merged. Thanks Ewan Crawford!)

Background: AdaptiveCpp generic SSCP JIT Compiler

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

Features

- Default compiler of AdaptiveCpp
- Single-source, single compiler pass (SSCP design)
- JIT compiles to: host ISA (any CPU), SPIR-V, PTX, amdgcn, MSL

Key Advantages

- Single-pass JIT compilation avoids parsing code multiple times for host and device.
- Binary portability due to JIT compilation
- Highly competitive performance; often outperforms vendor compilers
- Exploits runtime knowledge for additional performance optimization[1]

[1] Alpay et al. (2025): Adaptivity in AdaptiveCpp: Optimizing Performance by Leveraging Runtime Information During JIT-Compilation

Related Work (Selection)

Project	Approach	Focus
clang CUDA[2]	Alternative CUDA compiler	Same hardware
Coriander[3]	CUDA-on-OpenCL 1.2	Portability
chipStar[4]	HIP/CUDA on SPIR-V	OpenCL/L0 targets
ZLUDA[5]	Runtime CUDA recompilation	Binary portability
PCUDA (ours)	**Integration in AdaptiveCpp**	**Portability + SYCL interop**

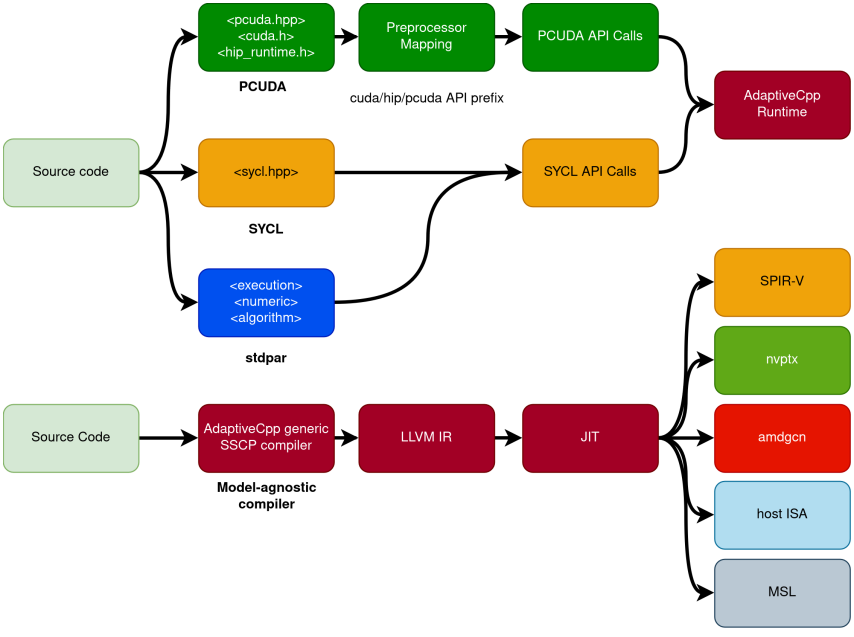
Differentiation: First compiler that enables seamless interoperability with SYCL at both source and runtime level while providing wide portability.

[2] Wu et al. (2016): gpucc: an open-source GPGPU compiler

[3] Perkins (2017): CUDA-on-CL: a compiler and runtime for running NVIDIA® CUDA™ C++11 applications on OpenCL™ 1.2 Devices

[4] Velesko et al. (2026): chipStar: Making HIP/CUDA applications cross-vendor portable by building on open standards

[5] Janik (2024): ZLUDA. `https://github.com/vosen/ZLUDA`


<pcuda.hpp>
<cuda.h>
<hip_runtime.h>
Preprocessor Mapping
PCUDA API Calls
PCUDA
cuda/hip/pcuda API prefix
AdaptiveCpp Runtime
Source code
<sycl.hpp>
SYCL API Calls
SYCL
<execution>
<numeric>
<algorithm>
stdpar
SPIR-V
nvptx
Source Code
AdaptiveCpp generic SSCP compiler
LLVM IR
JIT
amdgcn
host ISA
Model-agnostic compiler
MSL

SYCL and PCUDA use common compiler/runtime infrastructure.

But PCUDA is not (!!!) layered on top of SYCL.

- Remember goal: Fairer compiler comparisons between SYCL and CUDA
- $\rightarrow$ PCUDA should not be at a higher layer than SYCL!
- Both talk directly to compiler and runtime internals

Implementation: Kernel Launch Mechanisms

PCUDA supports three kernel launch mechanisms:

`<<<>>>`	`hipLaunchKernelGGL()`	`pcudaParallelFor()`
Original CUDA syntax	Function-based launch	SYCL-style launch

Implementation Strategy

All mechanisms mapped to internal `__pcudaPushCallConfiguration()` + kernel function call. This triggers kernel generation via `__global__` attribute.

Triple chevron syntax requires preprocessing step (`--acpp-pcuda-chevron-launch`).
Cannot use clang's CUDA frontend (not compatible with SSCP, not compatible with SYCL interop)

Implementation: Local Memory Handling

Two CUDA/HIP local memory patterns:

Static Local Memory

```
__shared__ int buffer[128];
```

LLVM transformation: Move to address space 3 (GPU) or thread-local memory (CPU).

Dynamic Local Memory

```
extern __shared__ int dynamic_mem[];
```

LLVM transformation: Replace with builtin returning pointer to configurable local memory region.

Result: Full CUDA/HIP local memory semantics preserved across all backends.

Implementation: SYCL-PCUDA Interoperability

Inside Kernels

All SYCL and PCUDA functionality works interchangeably:

```
__global__ void kernel(int* data) {
  auto nd_item = sycl::AdaptiveCpp_pcuda::this_nd_item<1>();
  int x = sycl::reduce_over_group(nd_item.get_group(), ...);
  x += data[threadIdx.x];
}
```

Runtime Interoperability

Objects translatable between models, e.g.

- Queues/streams, events, device memory pointers

Example: `sycl::AdaptiveCpp_pcuda::make_queue(stream)`

Use cases: Iterative porting, mixed CUDA/SYCL codebases.

Implementation: Limitations

- Not all APIs implemented yet (e.g. warp shuffles – compiler has builtins that users can call directly)
- CUDA/HIP is not standardized!
  - nvcc, nvc++, hipcc, clang speak different dialects due to compiler design (e.g. whether host/device attributes can overload, whether `warpSize` is constexpr)
  - Similarly, PCUDA speaks slightly different dialect – single pass compiler + JIT
  - CUDA/HIP apps **already today** need to take into account differences between compilers.
- Inline assembly is supported, but not recompiled $\rightarrow$ limits portability
- Vendor libraries which leverage inline assembly or assumptions about the specific CUDA/HIP “dialect” may not work
- Interop: Care must be taken when calling work-group collective functions (e.g. `__syncthreads()`) in SYCL `parallel_for(range<Dim>)`
  - Already today a problem in the SYCL ecosystem due to `work_item_queries` KHR extension

Experimental Setup

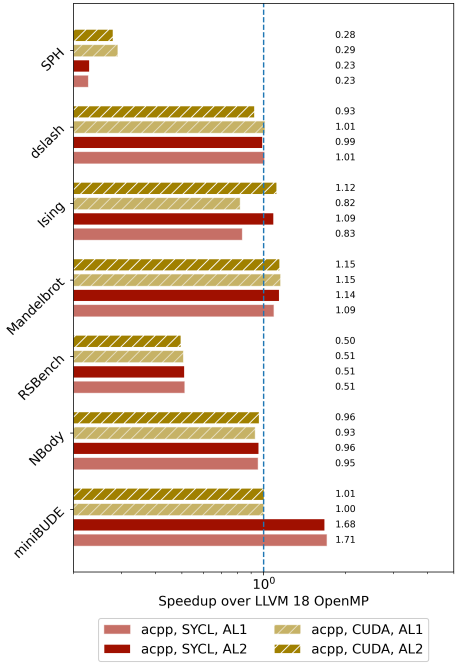
System	Hardware	Software Stack
1	AMD **Ryzen 9950X**	LLVM 18, Linux 6.12
2	2x AMD Epyc 7542 + 4x **NVIDIA A100**	CUDA 12.9, LLVM 20
3	Core i7-8700 + **Intel Arc B580**	oneAPI 2025.3, OpenCL 25.48
4	Xeon Platinum 8568Y+ + **AMD MI300X**	ROCm 6.4.1, LLVM 19

Benchmarks: miniBUDE, CloverLeaf, HeCBench suite (RSBench, SPH, nbody, Mandelbrot, dslash, ...)

Configuration: `-O3 -ffast-math` or equivalent

Evaluation: Performance

- Results: Performance normalized to native model compiled by vendor compiler (e.g. nvcc-compiled CUDA)
- Adaptivity Levels AL1/AL2: Aggressiveness of AdaptiveCpp JIT-time optimizations
- `acpp, SYCL`: Result of the SYCL port of the benchmark compiled with AdaptiveCpp
- `acpp, CUDA`: Result of the CUDA port of the benchmark compiled with AdaptiveCpp $\rightarrow$ PCUDA
- **Same compiler**, only different input code!

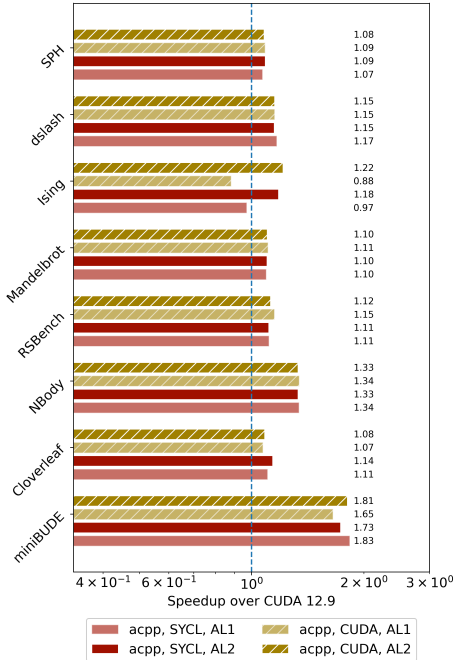


AMD Ryzen 9 9950X

- **Key Finding:** SYCL and PCUDA within 5% in most cases
- **miniBUDE:** SYCL 70% faster (uses local memory; CUDA version doesn't)
- **SPH:** Both underperform OpenMP (ports not CPU-optimized, e.g. group sizes)
- **RSBench:** 50% of OpenMP (JIT aggressive vectorization)

Takeaway

Performance differences primarily due to **input code differences**, not compiler or model.



NVIDIA A100

SYCL and CUDA (PCUDA) match within 5% at same adaptivity level

AdaptiveCpp speedup vs nvcc (geometric mean)

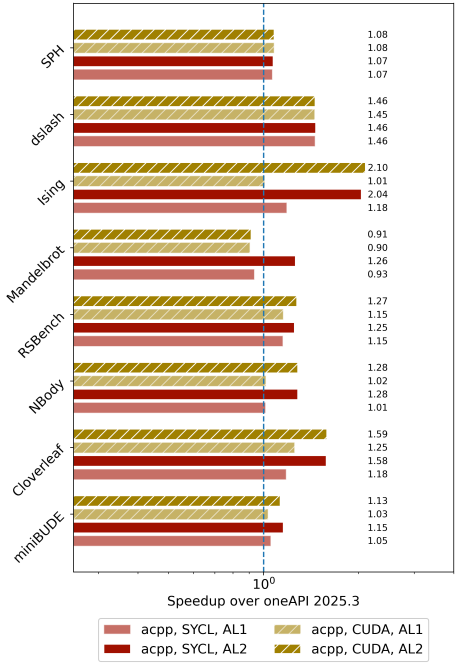
	AL1	AL2
SYCL	19%	21%
CUDA	16%	22%

(CUDA results: **Same input code as nvcc!**)

Key Insight

Geometric mean speedup confirms:

1. SYCL abstractions don't introduce inherent performance penalties
2. Speedup due to AdaptiveCpp compiler, not programming model
3. PCUDA enables fair CUDA-vs-SYCL comparison with same compiler



Intel Arc B580

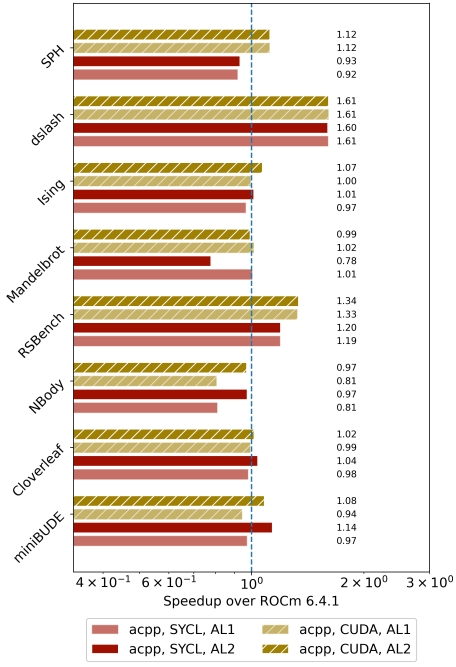
Speedup vs oneAPI icpx (geometric mean)

	AL1	AL2
SYCL	12%	36%
CUDA	10%	31%

SYCL and PCUDA perform within 5% at same adaptivity level.

Mandelbrot exception: Only SYCL benefits from AL2. CUDA passes params via pointer (obscures optimization), SYCL passes by value.

Confirmation: AdaptiveCpp competitive across Intel hardware.



AMD Instinct MI300X

AdaptiveCpp speedup vs hipcc (geometric mean)

	AL1	AL2
SYCL	4%	6%
CUDA	7%	13%

Notable results:

- dslash: 60% speedup (automatic pointer non-aliasing detection)
- SPH: PCUDA 1.12x vs SYCL 0.93x (vector alignment differences)

SYCL and PCUDA within 10% confirms minimal model overhead.

Performance Summary

	9950X	A100	B580	MI300X
SYCL AL1	0.78	1.19	1.12	1.04
PCUDA AL1	0.75	1.16	1.10	1.07
SYCL AL2	0.81	1.21	1.36	1.06
PCUDA AL2	0.77	1.22	1.31	1.13

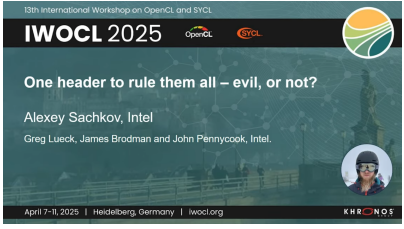
Table: *

Geometric mean speedup vs vendor compilers

Key Conclusions

1. SYCL and PCUDA performance closely matches across all platforms
2. Larger deviations traced to input code differences
3. AdaptiveCpp competitive on all tested hardware (4–36% speedup)

Compilation Time

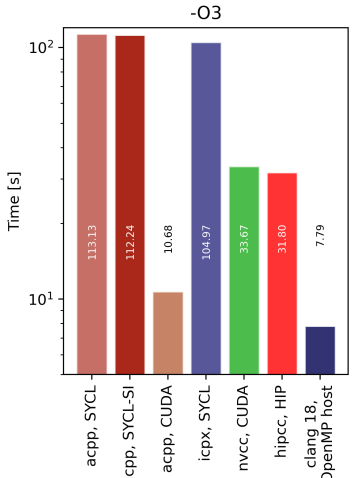


- Idea: Split up monolithic `<sycl.hpp>`
- → Proposed `sycl_khr_includes` KHR extension

Here:

- Compare compile times between SYCL and CUDA **with same compiler**
- AdaptiveCpp does not yet implement `sycl_khr_includes`; simulate by including internal headers
- Look at CloverLeaf (largest app among benchmarks)

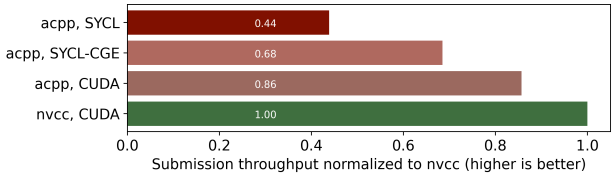
Compilation Time: CloverLeaf Benchmark



- PCUDA compiles 10x faster than SYCL (same compiler!)
- PCUDA 3x faster than nvcc (single-pass advantage)
- SYCL compile time issue is inherent to the model
- Splitting `sycl.hpp` yields negligible improvement → `includes` KHR extension does not always save us.
- Not due to frontend – middle-end/optimizer. SYCL headers exposes more implementation details?

Submission Overhead: Million Empty Kernels on A100

- Synthetic benchmark on NVIDIA A100
- Measure submission throughput only (not execution)
- Compare: SYCL, SYCL+CGE (acpp extension, omits mandatory events), PCUDA, nvcc



Analysis:

- SYCL event creation accounts for almost half of overhead
- Remaining overhead compounding from many smaller effects
- PCUDA achieves near-nvcc performance

Key Contributions

1. **First SYCL-interoperable CUDA/HIP compiler with portability to CPUs and GPUs** from multiple vendors
2. **Unified comparison framework:** Same compiler for CUDA and SYCL enables isolating model vs compiler effects
3. **Performance validation:** SYCL abstractions don't introduce inherent penalties vs CUDA.
   AdaptiveCpp is highly competitive due to compiler advantages, not because of SYCL.
4. **Compile time insight:** SYCL's compilation overhead is worse than previously thought ($\approx$3x versus nvcc, but $\approx$10x versus same compiler)
5. **Runtime overhead analysis:** Identified event creation as major SYCL overhead source

Conclusion

PCUDA enables:

- Direct compilation and execution of CUDA/HIP on heterogeneous hardware
- Seamless mixing with SYCL code
- Iterative or partial porting of CUDA projects
- Fair comparison between CUDA and SYCL programming models

Results:

- SYCL and PCUDA match performance within few percent
- AdaptiveCpp highly competitive: 4–36% speedup over vendor compilers
- 10x faster compilation than SYCL (10s vs 110s for CloverLeaf)
- PCUDA within 15% of nvcc for kernel submission throughput

PCUDA is publicly available as part of the AdaptiveCpp compiler project.

This work was partially supported by the SYCLops project, funded by the European Union's HE research and innovation programme under grant agreement No 101092877.